NOTE TO USERS

This reproduction is the best copy available.

UMI

University of Alberta

Monte Carlo Planning in RTS Games

by

Michael Chung

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

0-494-08039-6

Your file *Votre référence*
ISBN:
Our file *Notre reference*
ISBN:

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Monte Carlo simulations have been successfully used in classic turn-based games such as backgammon, bridge, poker, and Scrabble. In this thesis, we apply the ideas to the problem of planning in games with imperfect information, stochasticity, and simultaneous moves. The domain we consider is real-time strategy games. We present a framework — MCPlan — for Monte Carlo planning, identify its performance parameters, and analyze the results of an implementation in a capture-the-flag game.

Acknowledgements

I thank my supervisors, Michael Buro and Jonathan Schaeffer for their support and encouragement. I also thank Markus Enzenberger and Nathan Sturtevant for their valuable feedback. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

Contents

1 Introduction 1
- 1.1 Real–time Strategy Games . . . 1
  - 1.1.1 Warcraft . . . 1
  - 1.1.2 ORTS . . . 3
- 1.2 RTS AI . . . 4
- 1.3 Contributions of this Thesis . . . 7
- 1.4 Organization of Thesis . . . 7

2 Background 8
- 2.1 Data Structures . . . 8
  - 2.1.1 Influence Maps and Spatial Analysis . . . 8
  - 2.1.2 Resource Allocation Trees . . . 10
  - 2.1.3 Dependency Graphs . . . 10
- 2.2 Algorithms . . . 11
  - 2.2.1 Planning in Dynamic Worlds (RTS Games) . . . 11
  - 2.2.2 Adversarial Planning . . . 13
  - 2.2.3 Monte–Carlo Go . . . 14
  - 2.2.4 Simulation Based Planning . . . 16
  - 2.2.5 Random Map Generation for Strategy Games . . . 17
- 2.3 Analysis . . . 18
  - 2.3.1 Terrain Analysis . . . 18
  - 2.3.2 Multi-Tiered RTS Game AI . . . 19
- 2.4 Conclusions . . . 20

3 RTS Planning Methods 21
- 3.1 Expert Knowledge . . . 21
- 3.2 Monte–Carlo Planning . . . 22
  - 3.2.1 Top–Level Search . . . 23
  - 3.2.2 Abstraction . . . 24
  - 3.2.3 Evaluation Function . . . 24
  - 3.2.4 Plan Evaluation . . . 26

3.2.5 Comments 27
3.3 Capture the Flag 29
3.3.1 ORTS 30
3.3.2 CTF Game State Abstraction 31
3.3.3 Evaluation Function 31
3.3.4 Plan Generation 34
3.3.5 Plan Step Simulation 34
3.3.6 Other Issues 35
3.4 Summary 36

4 Experiments **37**
4.1 Experimental Design 37
4.1.1 Maps 38
4.1.2 Search Parameters 38
4.1.3 Players 39
4.1.4 Issues 39
4.2 Results 40
4.2.1 Increasing Number of Plans 40
4.2.2 Number of Units 41
4.2.3 Different Maps 41
4.2.4 Unbalanced Number of Units 42
4.2.5 Optimizing Max-Dist 43
4.2.6 Scripted Opponents 44
4.2.7 Run-Time for Experiments 46
4.3 Conclusions 46

5 Conclusions and Future Work **48**

Bibliography **50**

List of Figures

1.1 WarCraft 3 Base 2
1.2 WarCraft 3 AI Failing 3
1.3 ORTS 4

3.1 MCPlan: Top-level search 25
3.2 MCPlan: Plan evaluation 27
3.3 MCPlan: Plan simulation 28

4.1 Maps and unit starting positions 38
4.2 Increasing Number of Plans 40
4.3 Different Number of Units. MCPlan vs. Random 41
4.4 Different Maps. MCPlan vs. Random 42
4.5 Unbalanced Number of Units and Same AI 42
4.6 Less Men and Stronger AI vs. Random 43
4.7 Optimizing Max-Dist Parameter 44
4.8 MCPlan vs. Rush-the-Flag Opponent 45
4.9 MCPlan vs. Stand-Still Opponent 45

Chapter 1

Introduction

1.1 Real–time Strategy Games

Real–time strategy (RTS) games are popular commercial computer games involving a fight for domination between opposing armies. In these games, there is no notion of whose turn it is to move. Both players move at their own pace, even simultaneously; delays in moving will be quickly punished. Each side tries to acquire resources, use them to gain information and armaments, engage the enemy, and battle for victory. The games are typically fast–paced and involve both short–term and long–term strategies. The games are well–suited to Internet play. Many players prefer to play against human opponents over the Internet, rather than play against the usually limited abilities of the computer artificial intelligence (AI). Popular examples of RTS games include WarCraft [1] and Age of Empires [2].

1.1.1 Warcraft

The RTS game by Blizzard, WarCraft 3, offers 3D graphics and some nice artificial intelligence features. For example, some of the units are able to automatically cast spells during combat. This reduces the micromanagement typically handled by the player. For example, the "priest" units are able to automatically cast the "heal" spell on nearby injured friendly units. This is

Figure 1.1: WarCraft 3 Base

something that the player would have had to click really fast to accomplish in older RTS games, such as WarCraft 2.

Figure 1.1 is a screen-shot taken from WarCraft 3. The base in the figure is owned by the human player. The enemy base (not seen here) is owned by the computer AI player. The fog-of-war prevents each player from being able to see the enemy's forces or units until they come within visual range of a friendly building or unit. In the initial phase of an RTS game, each side builds up his army. The peons (workers) mine resources (gold and wood), which are then used by the peons to produce buildings such as the orc burrow (building in construction at the top) and the barracks (building in construction at the left). The barracks then produces military units such as the grunt (an orc foot soldier), which also requires resources (in this case, gold). When the orc burrow is manned by peons, it can attack, which provides some extra firepower for the defending player. In order to win the battle and the game, the attacking player must have enough forces to compensate for this defensive advantage.

Figure 1.2: WarCraft 3 AI Failing

In figure 1.2, the AI player is marching a lone worker into the player's army. Previously in this game, the AI's expansion town was destroyed by the player. While the AI's army has been defeated, it now repeatedly sends workers to rebuild the destroyed town, even though the player's army is still obviously waiting there. This poor play by the AI makes winning the game easy and boring.

1.1.2 ORTS

As commercial game companies do not provide their source code to the public, it is difficult to test new research ideas. The Open Real Time Strategy (ORTS) platform developed by Michael Buro is an open source real-time-strategy game [13]. The ORTS system is used to implement a simplified RTS game for testing our ideas.

While still a work in progress, ORTS already supports most of the features of RTS games, including 3-D graphics, fog-of-war, and a minimap (miniature

Figure 1.3: ORTS

view of the map). Figure 1.3 shows a sample screen–shot.

1.2 RTS AI

The AI in RTS games is usually achieved using scripting. Over the past few years, scripting has become the most popular representation used for expressing character behaviours. Scripting, however, has serious limitations. It requires human experts to define, write, and test the scripts comprised of 10s, even 100s, of thousands of lines of code. Further, the AI can only do what it is scripted to do, resulting in predictable and inflexible play. The general level of play of RTS AI players is weak. To enable the AI to be competitive, game designers often give the AI access to information that it should not have or increase its resource flow (i.e., they cheat).

Success in RTS games revolves around planning in various areas such as resource allocation, force deployment, and battle tactics. The planning tasks

in an RTS game can be divided into three areas, representing different levels of abstraction:

1. **Unit control (unit micromanagement).** At the lowest level is the individual unit. It has a default behaviour, but the player can override it. For example, a player may micromanage units to improve their performance in battle by focusing fire to kill off individual enemy units.

2. **Tactical planning (mid–level combat planning).** At this level, the player decides how to conduct an attack on an enemy position. For example, it may be possible to gain an advantage by splitting up into two groups and simultaneously attacking from two sides.

3. **Strategic planning (high–level planning).** This includes common high–level decisions such as when to build up the army, what units to build, when to attack, what to upgrade, and how to expand into areas with more resources.

In addition, there are other non–strategic planning issues that need to be addressed, such as pathfinding.

Unit control problems can often be handled by simple reactive systems implemented as list of rules, finite state machines, neural networks, etc. Tactical and strategic planning problems are more complicated. They are real–time planning problems with many states to consider in the absence of perfect information. It is apparent that current commercial RTS games deal with this in a simple manner. All of the AI's strategies in the major RTS games are scripted. While the scripts can be quite complex, with many random events and conditional statements, all the strategies are still predefined beforehand. This limitation results in AI players that are predictable and thus easily beaten. For casual players, this might be fun at first, but there is no re–playability. It is just no fun to beat an AI player the same way over and over again.

In RTS games, there are often hundreds of units that can all move at the same time. RTS games are fast–paced, and the computer player must be able to make decisions at the same speed as a human player. At any point in time, there are many possible actions that can be taken. Human players are able to quickly decide which actions are reasonable, but current state–of–the–art AI players cannot. In addition, players are faced with imperfect information, i.e. partial observability of the game state. For instance, the location of enemy forces is initially unknown. It is up to the players to scout to gather intelligence, and act accordingly based on their available information. This is unlike the classical games such as chess, where the state is always completely known to both players. For these reasons, heuristic search by itself is not enough to reason effectively in an RTS game. For planning purposes, it is simply infeasible for the AI to think in terms of individual actions. Is there a better way?

Monte Carlo simulations have the advantage of simplicity, reducing the amount of expert knowledge required to achieve high performance. They have been successfully used in games with imperfect information and/or stochastic elements such as backgammon [31], bridge [17], poker [10], and Scrabble [27]. Recently, this approach has been tried in two-player perfect-information games with some success (Go [12]). A framework for using simulations in a game–playing program is discussed in [26], and the subtleties of getting the best results with the smallest sample sizes is discussed in [28].

Can Monte Carlo simulations be used for planning in RTS games? If so, then the advantages are obvious. Using simulations would reduce the reliance on scripting, resulting in substantial savings in program development time. As well, the simulations will have no or limited expert bias, allowing the simulations to explore possibilities not covered by expert scripting. The AI would no longer be as predictable. The result could be a stronger AI for RTS games and a richer gaming experience. The applications of RTS AI are by

no means limited to creating opponents to entertain game players. There are many similar planning problems outside of games, such as military planning and robotics control.

1.3 Contributions of this Thesis

The contributions in this work are as follows:

1. Design of a Monte Carlo search engine for planning (MCPlan) in domains with imperfect information, stochasticity, and simultaneous moves.
2. Implementation of the MCPlan algorithm for decision making in a real–time capture–the–flag game.
3. Characterization of MCPlan performance parameters.

1.4 Organization of Thesis

Chapter 2 provides an overview of related research, including the current state–of–the–art in RTS game AI. Some ideas such as the Monte–Carlo Go algorithm are presented, which could be applied to RTS games. Chapter 3 describes the MCPlan algorithm and the parameters that influence its performance. Chapter 3 also discusses the implementation of MCPlan in a real–time capture–the–flag game built on top of the free ORTS RTS game engine [13]. Chapter 4 presents experimental results. Chapter 5 presents our conclusions and remarks on future work in this area.

Chapter 2

Background

This chapter provides an overview of planning in RTS games. The current state-of-the-art algorithms and data structures are described. The Monte Carlo Go algorithm is presented, an idea which could be applied to RTS games.

2.1 Data Structures

2.1.1 Influence Maps and Spatial Analysis

While some types of game behavior can be scripted as action sequences, this approach breaks down during player interaction. AIs for highly interactive games need to be able to reason about their environment. To do this, a high level of abstraction is required [33].

Designers can place hints in the game world such as the locations of the important choke–points [22]. However, this only works for static environments [33]. RTS games, in contrast, are dynamic. For example, in WarCraft, trees can be cut down as part of the game, possibly creating new choke–points and opening up old choke–points.

Influence mapping is a technique used to perform dynamic spatial reasoning. These 2D arrays are abstractions of the original RTS map, which aid in both tactical and strategic planning. Influence maps are commonly used by game AIs for tactical assessment. They can indicate where friendly and

enemy forces are located, where battles have occurred, areas of control, unexplored areas, and areas likely to be attacked. Using influence maps, the AI can also infer which areas are secured, areas of enemy weakness (such as rear and flanks), good locations for resting and healing, good locations for ambushes, areas that should be defended, choke–points, etc [32].

Influence maps can be generated as follows. Each cell in the influence map is initialized to its influence value, which could be combat strength, area visibility, or any other feature. The value is blurred to nearby cells using an arbitrary falloff rule, such that the influence of the cell decreases with distance. This blurring reflects that units could quickly move to or attack nearby locations. In practice, influence map cells should be large (about 10x10 or 20x20 tiles) [32]. This provides different levels of abstraction for higher level planning.

Spatial databases allow AI to reason about their environment in other ways [33]. Essentially an extension of the influence mapping idea, spatial databases consist of multiple layers of 2D grids, overlaid on the game world. Each layer describes a different aspect of the environment. For example [33]:

- **Openness layer**: proximity to obstacles
- **Cover layer**: locations where agents can take cover
- **Area search layer**: for searching area for intruders

In [33], a few of the typical layers are described in more detail. These layers can be combined to perform different types of reasoning. In other words, a "desirability layer" can be formed by combining any of the existing layers. Also, sharing spatial databases between agents can help in coordination [33].

2.1.2 Resource Allocation Trees

The resource allocation tree helps the AI player decide how to allocate resources. This is another data structure which provides abstracted information about the current RTS game state. This information aids in strategic planning.

In a resource allocation tree, a player's units and other assets are divided into a tree structure. Under the root node are all the player's current strategic assets. At each node, the player's assets are divided based on functional purpose, such as military, intelligence, and economy. As we go down the tree the nodes become more and more specific, until the leaf nodes, which are specific unit types. The leaf nodes contain values that should be dependent on the branch. Nodes under the military branch should assign values based on unit strength. Nodes under the intelligence branch should assign values based on unit speed and sight radius. Of course, this is made more complicated in games with many different types of resources [32].

This tree is mainly used to decide what new units to build, and what to do with existing units. A combat balancing table [32] is used to look up a unit's effectiveness versus any other unit. This allows the AI player to build units that counter the units it thinks that the enemy has.

2.1.3 Dependency Graphs

Dependency graphs show a dependency of assets. This covers the concepts of technology-trees and building-trees, which indicate the requirements of each building or unit. For example, in WarCraft 3, a barracks is required to build grunts. There are also abstract dependencies in the graph, such as the fact that peons are required for gathering gold and lumber. Resource dependencies are also considered, such as how much gold and lumber are required to build a barracks. Another type of dependency is a support dependency. An example of this is how an orc burrow requires peons inside it to fire at nearby enemies.

Without the peon the burrow cannot attack [32].

Each node in the graph contains useful data such as the estimated number and value of units or buildings of that type, owned by the player. Units and buildings under construction are considered as well. This data is similar to the information in the resource allocation tree nodes. However, while the resource allocation tree only tracks current assets, the dependency graph tracks all possible assets.

Dependency graphs can be used to perform strategic inference. For example, if we see an enemy grunt, then we know that the enemy has a barracks. Similarly, if we see a barracks, then we can expect to see enemy grunts later. This can provide us with statistical information about the current RTS game state. The AI player might use this information to decide, for example, to build more base defenses to protect itself from possible grunt attacks.

The graph is useful for identifying weak nodes. For example, if the enemy only has one barracks, then the AI player would prefer to attack the barracks and destroy it. A smart AI player might build two barracks, in case the enemy attempts the same tactic.

2.2 Algorithms

2.2.1 Planning in Dynamic Worlds (RTS Games)

Goal-directed behavior is widely used in computer games. It establishes a sense of purpose for the agents, and increases the believability of their actions [14]. The agents accomplish their goals by executing plans, which could be generated in a variety of ways.

Traditional game agents are reactive. They respond to situations using hard-coded responses, rather than planning at run-time. The responses are often simple, such as returning fire when an enemy is sighted. There are many problems with reactive approaches. They rely heavily on developers to

think of all possible situations that the agent may encounter, and to program the proper responses. These pre-canned solutions are becoming more difficult to manage as game worlds are becoming more complex. Also, cooperative behavior is extremely difficult [34].

Classical planning (such as STRIPS [15]) formulates the entire plan before execution, assuming implicitly that the world does not change while planning and executing the plan. However in a game the world is constantly changing. Also, classical planning can be a very expensive calculation which may not be suitable in a game environment due to time and CPU resource constraints.

Orkin describes a decision-making architecture, Goal-Oriented Action Planning (GOAP), which allows characters to decide not only what to do, but how to do it [23]. It results in characters that are less predictable, and able to adapt to new situations. GOAP makes use of the A* algorithm to formulate plans, as is done by many classical planners. It is interesting to note that although Orkin uses the game "No One Lives Forever 2" (NOLF2) to provide examples where GOAP may help, GOAP is not actually implemented in NOLF2 [23]. Unfortunately, GOAP is essentially the same as classical planning. While it is a step in the right direction, and certainly better than reactive behavior, it may not scale up well to RTS games where there are many units.

Some plans use a high level of abstraction, which is insufficient for controlling an agent. Low-level plans have too many steps so that a solution cannot be found in a reasonable amount of time (with basic planning techniques). Some current practical planners use hierarchical decomposition or hierarchical task network (HTN) planning. A plan is formulated at an abstract level, with abstract operators. The abstract operators are decomposed into lower level steps. The decompositions of the abstract operators can be stored in a library and retrieved as needed [25].

HTN planners are more effective than classical planners in a dynamic game environment. It's noted that the performance is better in HTN planning (linear

in ideal case, instead of exponential)[34]. Partial re–planning can be accomplished when using HTN planning. Invalidated plans can be adapted (at a lower level) instead of completely re–planning [34]. Also, agent cooperation is more easily possible with HTN planning, using synchronization actions for example. There is a question of what abstraction level to use for the coordination. There is a trade–off between ease and crispness of coordination.

An example of HTN planning is seen in the game Full Spectrum Command [14]. The implementation uses composite tasks, which are composed of other composite tasks, or simple tasks. The simple tasks are composed of actions. This planning technology is also demonstrated by Soar technology and Quake bots [3].

2.2.2 Adversarial Planning

Willmott has recently made some progress in adversarial planning in Go [35] which is a game that, while turn-based, is more difficult than RTS in some ways. For example, evaluating positions in Go is difficult. In RTS it is pretty easy, as material (total value of units, buildings and resources) is a pretty good measure of the value of a position. The slightest different in a Go position can mean the difference between "life and death". In RTS games, small changes in the state usually do not affect the outcome of the game.

Willmott's adversarial planning is based on HTN planning. Like other planning techniques, it is goal-directed. Adversarial planning assumes that there are two opposing agents, which attempt to satisfy their own goals while stopping the goals of the opponent. The plan tree is searched, considering the interacting plans of both agents. When one agent achieves all of its goals, backtracking occurs, and the next branch is explored. The application of this framework to Go resulted in the implementation of GOBI, which has proven successful in solving many beginner-level Go life-and-death problems [35].

A problem with this approach is that searching for life-and-death is not

the same as trying to play an RTS game well. Most of the time we will not be able to plan ahead far enough to see the end of the game, so we need to set realistically achievable sub-goals. Intelligent generation of goals for input to the adversarial planner is a separate problem, although, it is also interesting. Also, this approach does not handle the timed actions, parallel actions, and uncertainty in the RTS domain. Like Alpha-Beta, it assumes that the two players take turns. While adversarial planning techniques such as that used in GOBI succeeds in some cases where Alpha-Beta fails, it still does not address the main issues of RTS adversarial planning problems.

2.2.3 Monte–Carlo Go

Bouzy and Helmstetter describe the computer Go programs OLGA (by Bouzy) and OLEG (by Helmstetter)[12]. The basic idea behind both programs is a Monte–Carlo approach, where moves are evaluated based on the outcome of a large number of random games played from the current position in the game. The random games are completely played out and then scored. Each random game begins with the move that is being evaluated. Each move after that is randomly selected from all possible moves, excluding moves that are suicidal. The mean of scores of the random games is used as the evaluation for each first move. This is simpler than Bruegmann's approach (GOBBLE) [4], and based on Abramson's method [9].

This approach works well in Go where a full global game tree search is not practical in most cases. Domain dependent move generators can produce good moves, but always contains errors. This Monte–Carlo approach provides a sort of verification of the strength of a move [12].

Speed is a factor in making this approach viable: on a 2 GHz computer, OLGA plays 7,000 random 9x9 games per second, and OLEG plays 10,000. It is noted that a 20% speed increase results in a 10% improvement in precision, which is not very significant. However, a 10X speed increase would result

in a 3X increase in statistical confidence, which makes a first pass of code optimizations worthwhile [12].

In addition, the following ideas for possible enhancements have been tested within the Monte-Carlo framework:

- **all moves as first heuristic**: making statistics for all moves of the random game, rather than just for the first move. The idea is that in a random go game, any of the friendly moves could have been played as the first move, so as a speed-up, we can evaluate all the moves at the same time. There are, of course, cases where this is not reliable.
- **progressive pruning**: moves with evaluation too low compared to the best move are pruned.
- **simulated annealing**: in a random game, instead of selecting each move with equal probability, simulated annealing is used to determine the probability of each move, based on its current evaluation.
- **temperature**: variation of simulated annealing, where temperature is kept constant. The result is that the probability of selecting each move is a simple function of its current evaluation.
- **depth-two tree search**: perform mini-max search to depth two and evaluate the resulting positions using random games.

The resulting programs were somewhat successful against respectable Go programs such as INDIGO (Bouzy, 2002) and GNUGO (Bump, 2003) in 9x9 Go [12] and 19x19 Go [11].

This approach scales nicely with CPU speed and is trivially parallel — unlike alpha–beta. It is concluded that as computers become more powerful, this type of Monte–Carlo approach will become more worthy of consideration [12].

2.2.4 Simulation Based Planning

The term "Simulation-Based Planning (SBP)" seems to have been introduced by Lee and Fishwick [21]. SBP has been successfully applied to route planning, military mission planning and controlling a truck depot [20].

The idea is that given a set of possible plans, each plan is executed on a simulator, with varying levels of abstraction. Given the result of the simulation, an objective function determines the value of each plan based on the simulation results, so the planning agent can act accordingly. The level of abstraction is adapted to the available planning time. Also, depending on planning time available, planning simulations could be repeated many times to generate statistically relevant results. This naturally handles malicious environments, adversarial agents, uncertainty and randomness. For situations where immediate response is required, and there is no time for any simulation at all, the planner reverts to a "reactive behaviour mode" [20].

SBP sort of lends itself to RTS planning. All RTS games are simulations. Stochastic elements are handled in a reasonable way. Intuitively, it should simulate the correct result most of the time. Plans such as "attack" could be evaluated by the planning simulator, while plans such as "defend" would be handled reactively.

The main drawback of SBP is that it does not effectively anticipate the adversary's actions. In the planning simulations, the adversaries are usually assumed to be quite simple, in order to be able to run the simulation efficiently [21]. In RTS games, the skill of the opponent will greatly affect the outcome of any battle. It is naive to assume that the opponent will behave simply. While it is claimed that "multimodelling" can be used to handle multiagent adversarial planning in real-time [20], this claim seems a little optimistic and requires further investigation.

Intuitively, SBP would seem to be very computationally expensive. In a

commercial RTS game, computation time is at a premium. However, with the proper abstract model, the planning simulator could potentially be extremely fast. For example, it would quickly compute the outcome of battles, using the abstracted data found in influence maps and resource allocation trees. Definitely for dedicated machines running RTS AI for research or competition purposes, simulation may be a good way of evaluating plans generated by other, more time-efficient planning approaches. With the proper abstractions and optimizations, it may yet be practical.

Also, it might be better if, rather than simply evaluating plans, the planning simulator returns an analysis of the entire simulation. In other words, show what went wrong with the plan. This would help identify the flaws in the plan, so that the high level planner could make the appropriate modifications to the plan.

2.2.5 Random Map Generation for Strategy Games

Shoemaker [29] describes the challenges and techniques involved in random map generation for strategy games (as implemented in the RTS Game Empire Earth by Sierra [5]). In order to perform unbiased experiments, the random maps generated must be roughly fair to both players.

The problem is to generate a random map for a tile–based RTS game. The land and resources for each player must be about equal. There must be no strategic advantage to either player. The map must be suitable for both hard–core RTS gamers as well as casual players. The process should be relatively quick, and should take advantage of available memory (other processes will not be running during map generation). The generator should be able to create different map types, such as small islands, etc. [29].

Shoemaker gives an overview of the process. The players start on a blank map, with teams adjacent and enemies somewhat symmetrically opposed. The area starts as all water, and the land is grown in clumps, to appear natural.

Flat land areas are created for the players, as well as some areas for later expansions. A height map is generated, with realistic elevation. The results are combined, with some restrictions (filters) applied. The map is processed for cliffs and other distinctive geographic features (terrain analysis). The resources are placed in rings around the players and finally the units are placed [29].

A test application was developed to show the map before the random map generator was integrated into the game engine. Scripts are used to help design the maps. These lengthy, complex scripts were created by the programmers and tweaked by the designers [29].

Grimani describes the wall-building aspect of a random map generator [18].

2.3 Analysis

2.3.1 Terrain Analysis

Terrain analysis tools are used in RTS games to provide abstract information about the terrain. In Age of Empires, for instance, terrain analysis tools used include pathfinding, influence maps, and area decomposition. A common use of terrain analysis is to provide the AI player with information about which areas are reachable. It was implemented using pathfinding, which was always accurate, but slow [6].

Area decomposition is the division of the map into areas. Terrain analysis can be useful for determining area connectivity. Areas can be modified by a scenario editor. This helps the performance of high-level pathfinding. It's possible to track data on the different areas: unit counts, resources and connections to other areas [6].

For abstraction and planning purposes, having the map divided into areas is helpful. Terrain analysis can be used to identify areas with the best resources. More abstract information about each area can be inferred, such as whether

an area is a good place to build an expansion.

2.3.2 Multi-Tiered RTS Game AI

RTS AI has to handle many units. Doing this on an individual unit level is impractical. Thus there must be some sort of hierarchical reasoning or planning.

Kent [19] suggests that this problem has already been solved for thousands of years by the military. The hierarchical chain of command has allowed army generals to control thousands of troops.

Soldiers are grouped into squads in many games. This makes it easier to perform tactics such as an attack on a target from two directions. Instead of planning for scores of units, the AI only has to plan for a couple squads (or companies).

The AI is divided into tiers. The highest level is the computer player AI, and the lowest level is the soldier AI. While all RTS games have these two layers, the introduction of intermediate layers reduces the workload for the highest and lowest level layers. AIs communicate with superior and subordinate AIs. They also communicate with other AIs of the same layer. The messages include orders and feedback messages.

Kent lists the following possible AI tiers or layers: Soldier AI, Squad AI, Platoon AI, Company AI, Brigade AI, Division AI, Army AI, Computer Player AI [19]. The exact number of tiers is not important.

All RTS games already have the rudimentary soldier AI. Its job is to follow orders and stay alive. The soldier AI reports to the squad AI about things such as new enemies spotted. It selects and engages targets, and path–finds using way–points given by the squad AI.

The squad AI is also implemented in many games. It takes orders from the next higher AI (the platoon AI) and distributes the orders to the squad members. Feedback from the soldiers is evaluated and reported to the platoon

AI.

The platoon, and each higher level, is similar to the squad AI. Each higher level has fewer details, and is more concerned with abstract concepts such as offensive ratings. For example, the platoon AI reasons about the formation of its squads, rather than the locations of each soldier.

Each AI tier has a different view of the game map, with the level of detail suitable for its planning tasks. While the soldier AI needs details about the environment, the highest level AI only needs to know about the abstract features such as locations of strategic importance [19].

Similarly, Ramsey [24] divides the RTS game AI into the following tiers: strategic intelligence, operational intelligence, tactical intelligence, and individual unit. The strategic intelligence handles the grand strategy, such as which city to capture, and when to capture the city. The operational intelligence handles the details of how best to accomplish these objectives. The tactical intelligence handles small-scale interactions, such as scouting a battlefield or capturing an enemy city [24].

2.4 Conclusions

The state of the art in AI for RTS leaves much to be desired. Besides being repetitive and too easy to beat for many players, it is also a huge burden on the game developers. As many of the plans are scripted and game specific, the AI programmers have little opportunity for code reuse. Each scripted plan needs to be written by hand specifically for that game and then tested to make sure it handles each game situation correctly. Both the writing and testing of scripts is labour intensive and time consuming.

While some developers are certainly looking at adding planning AI to their games, many classical planning techniques simply do not scale up to the problems that occur in real-time strategy games.

Chapter 3

RTS Planning Methods

This chapter describes solution methods to the RTS planning problem. First, we review what current commercial RTS games use: expert–knowledge–based systems. We will then introduce and describe an approach based on Monte–Carlo sampling, simulation and replanning, which is much less reliant on human expert knowledge.

3.1 Expert Knowledge

Scripted solutions are used in most if not all commercial RTS games, including StarCraft, WarCraft 3 and Ages of Empires 2. While they can be quite effective, they are lacking in several ways. First, they do not fully address the issue of player interaction. In many cases, they may seem to, because of the effort the developers and script writers put into predicting and handling many possible scenarios, and hard-coding the responses. While this leads to the desired behavior in the commonly occurring cases, the AI is unable to adapt to new situations. Every time the game changes, as it does often in the development of any new RTS title, the scripts must be updated and re-tested. This huge burden on the AI programmers and testers is a downside to expert-knowledge-based systems. Another downside is that in order to script the AI, an expert is required. For new games, there is often no expert available on

the development team, besides the designers of the game (who probably have better things to do than help script the AI).

In games such as WarCraft 3[1], the AI player has a fixed set of rules that dictate how it plays the game. For example, it starts out the game by sending peons to the mine, building an orc burrow and a barracks, and producing three grunts. It does this every game, with no variation. So another downside of rule-based systems is the lack of excitement caused by this repetition. Other games have a slightly better solution. They have a list of many scripts, and randomly choose among them. This seems to give the illusion that the AI player is adapting and trying new things, when in reality it is simply choosing from the large list of scripted plans. Randomness hides repetitiveness, and can create the illusion of intelligence.

Other examples of expert knowledge systems include rule–based systems such as Soar [3]. While more sophisticated than what is (probably) in commercial RTS games, it is still limited by the fact that the rules need to be written and maintained by human experts.

3.2 Monte–Carlo Planning

Adversarial planning in imperfect information games with a large number of move alternatives, stochasticity, and many hidden state attributes is very challenging. Further complicating the issue is that many games are played with more than two players. As a result, applying traditional game–tree search algorithms designed for perfect information games that act on the raw state representation is infeasible. One way to make look–ahead search more effective is to abstract the state space. An approach to deal with imperfect information scenarios is sampling. The technique we present here combines both ideas.

Monte–Carlo sampling has been effective in stochastic and imperfect information games with alternating moves, such as bridge [17], poker [10], and

Scrabble [27]. Here, we want to apply this technique to the problem of high–level strategic planning in RTS games. Applying it to lower level planning is possible as well. The impact of an individual move — such as a unit moving one square — requires a very deep search to see the consequences of the move. Doing the search at a higher level of abstraction, where the execution of a plan becomes a single "move", allows the program to envision the consequences of actions much further into the future (see Section 3.2.2).

Monte–Carlo planning (MCPlan) does a stochastic sample of the possible plans for a player and selects the plan to execute that has the highest statistical outcome. The advantage of this approach is that it reduces the amount of expert–defined knowledge required. For example, Full Spectrum Command [7] requires extensive military–strategist–defined plans that the program uses — essentially forming an expert system. Each plan has to be fully specified, including identifying the scenarios when the plan is applicable, anticipating all possible opponent reactions, and the consequences of those reactions. It is difficult to get an expert's time to define the plans in precise detail, and more difficult to invest the time to analyze them to identify weaknesses, omissions, exceptions, etc. MCPlan assumes the existence of a few basic plans (e.g. explore, attack, move towards a goal) which are application dependent, and then uses sampling to evaluate them. The search can sample the plans with different parameters (e.g. where to attack, where to explore) and sequences of plans—for both sides. In this section, we describe MCPlan in an application–independent manner, leaving the application–dependent nuances of the algorithm to Section 3.3.

3.2.1 Top–Level Search

The basic high–level view of MCPlan is as follows, with a more formal description given in Figure 3.1:

1. Randomly generate a plan for the AI player.

2. Simulate randomly–generated plans for both players and execute them, evaluate the game state at the end of the sequence, and compute how well the selected plan for the AI player seems to be doing (`evaluate_plan`, Section 3.2.3).

3. Record the result of executing the plan for the AI player.

4. Repeat the above as often as possible given the resource constraints.

5. Choose the plan for the AI player that has the best statistical result.

The variables and routines used in Figure 3.1 are described in subsequent subsections.

The top–level of the algorithm is a search through the generated plans, looking for the one with the highest evaluation. The problem then becomes how best to generate and evaluate the plans.

3.2.2 Abstraction

Abstraction is necessary to produce a useful result and maintain an acceptable run–time. Although this work is discussed in the context of high–level plans, the implementor is free to choose an appropriate level of abstraction, even at the level of unit control, if desired. However, since MCPlan relies on the power of statistical sampling, many data points are usually needed to get a statistically meaningful result. For best performance, it is important that the abstraction level be chosen to make the searches fast and informative.

In Figure 3.1, `State` represents an abstraction of the current game state. The level of abstraction is arbitrary, and in simple domains it may even be the full state.

3.2.3 Evaluation Function

As in traditional game–playing algorithms, at the end of a move sequence an evaluation function is called to assess how good or bad the state is for the

```
// Plan: contains details about the plan
// For example, a list of actions to take
class Plan {
  // returns true if no actions remaining in the plan
  bool is_completed();
  // [...] (domain specific)
};

// State: AI's knowledge of the state of the world
class State {
  // return evaluation of the current state
  // (domain specific implementation)
  float eval();
  // [...] (domain specific)
};

// MCPlan Top-Level
Plan MCPlan(
  State state,   // current state of the world
  int num_plans, // number of plans to evaluate
  int num_sims,  // simulations per evaluation
  int max_t)     // max time steps per simulation
{
  float best_val = -infinity;
  Plan best_plan;

  for (int i = 0; i < num_plans; i++) {
    // generate plan using (domain-specific) plan generator
    Plan plan = generate_plan(state);
    // evaluate using the number of simulations specified
    float val = evaluate_plan(plan, state, num_sims, max_t);
    // keep plan with the best evaluation
    if (val > best_val) {
      best_plan = plan;
      best_val = val;
    }
  }
  return best_plan;
}
```

Figure 3.1: MCPlan: Top-level search

side to move. This typically requires expert knowledge although the weight or importance of each piece of expert knowledge can be evaluated automatically, for example by using temporal difference learning [30]. For most application domains, including RTS games, there is no easy way around this dependence on an expert. Note that, unlike scripted AI which requires a precise specification and extensive testing to identify omissions, evaluation functions need only give a heuristic value.

3.2.4 Plan Evaluation

Before we describe the search algorithm in more detail, let us define the key search parameters. These are variables that may be adjusted to modify the quality of the search, as well as the run–time required. The meaning of these parameters will become more clear as the search algorithm is described.

1. `max_t`: the maximum time, in steps or moves, to look ahead when performing the simulation–based evaluation.

2. `num_plans`: the total number of plans to randomly generate and evaluate at the top–level.

3. `num_sims`: the number of move sequences to be considered for each plan.

The `evaluate_plan()` function is shown in Figure 3.2. Each plan is evaluated `num_sims` times. A plan is evaluated using `simulate_plan()` by executing a series of plans for both sides and then using an evaluation function to assess the resulting state. In the pseudo–code given, the value of a plan is the minimum of the sample values (a pessimistic assessment). Other metrics are possible, such as taking the maximum over all samples, the average of the samples, or a function of the distribution of values. The best metric is domain–specific, and could depends on the AI play–style desired. A pessimistic assessment results in defensive play, with fewer mistakes. Also, in the presented formulation of

```
// Evaluate Plan Function. Takes minimum of num_sims
// plan simulations (pessimistic)
float evaluate_plan(Plan plan, State state,
                    int num_sims, int max_t)
{
  float min = infinity;
  for (int i = 0; i < num_sims; i++) {
    float val = simulate_plan(plan, state, max_t);
    if (val < min) min = val;
  }
  return min;
}
```

Figure 3.2: MCPlan: Plan evaluation

MCPlan information about the plan chosen by the player is implicitly leaked to the opponent. This turns a possible imperfect information scenario into one of perfect information leading to known problems [16]. We will address this problem in future work. Here, we restrict ourselves to a simple form which nevertheless may be adequate for many applications. Each data point for a plan evaluation is done using `simulate_plan()`. A "game" consists of both sides selecting a plan and then executing it. This is repeated until time runs out. The resulting state of the game is assessed using the evaluation function. Note that opponent plans can cause interaction; how this is handled is application dependent and it is discussed in Section 3.3. The `evaluate_plan()` function calls `simulate_plan()` `num_sims` times, and takes the minimum value. The `simulate_plan()` function is shown in Figure 3.3.

3.2.5 Comments

MCPlan is similar to the stochastic sampling techniques used for other games. The fundamental difference — besides obvious semantic ones such as not requiring players to alternate moves — is that the "moves" can be executed at an abstract level. Abstraction is key to getting the depths of search needed

```
// Simulate Plan. Perform a single simulation with the given
// plan and return the resulting state's evaluation.
float simulate_plan(Plan plan, State state, int max_t) {
  State bd_think = state;
  Plan plan_think = plan;

  // generate a plan for the opponent (domain specific)
  Plan opponent_plan = generate_opponent_plan(state);

  while (true) {
    // simulate a single time step in the world
    // (domain specific)
    simulate_plan_step(plan_think, opponent_plan, bd_think);

    // check if maximum time steps has been simulated
    if (--max_t <= 0) break;

    // check if plan has been completed
    if (plan_think.is_completed()) break;

    // check if the opponent's plan has been completed
    if (opponent_plan.is_completed()) {
      // if so, generate a new opponent plan
      opponent_plan = generate_opponent_plan(bd_think);
    }
  }
  return bd_think.eval();
}
```

Figure 3.3: MCPlan: Plan simulation

to have long-range vision in RTS games. MCPlan lessens the dependence on expert-defined knowledge and scripts. Expert knowledge is needed in two places:

1. Plan definitions. A plan can be as simple or as detailed as one wants. In our experience, using plan *building blocks* is an effective technique. Detailed plans are usually composed of a series of repeated high-level actions. By giving MCPlan these building block actions and allowing it to combine them in random ways, the program can exhibit subtle and creative behaviour.

2. Evaluation function. Constructing accurate evaluation functions for non-trivial domains requires expert knowledge. In the presence of look-ahead search, however, the quality requirements can often be lessened by considering the well-known trade-off between search and knowledge. A good example is chess evaluation functions, which — combined with deep search — lead to World-class performance, in spite of the fact that the used features have been created by programmers rather than chess grandmasters. Because RTS games have much in common with classical games, we expect a similar relationship between evaluation quality and search effort in this domain, thus mitigating the dependency on domain experts.

3.3 Capture the Flag

Commercial RTS games are complex. There are many different variations, some involving many RTS game elements such as resource gathering, technology trees, and more. To more thoroughly evaluate our RTS planners, we limit our tests to a single RTS scenario, capture-the-flag (CTF). Our CTF game takes place on a symmetric map, with vertical and horizontal walls. The two forces start at opposing ends of the map. Initially the enemy locations are unknown.

The enemy flag's location is known — otherwise much initial exploration

would be required. This is consistent with most commercial RTS games, where the same maps are used repeatedly, and the possible enemy locations are known in advance.

The rules of our CTF game are relatively simple. Each side starts with a small fixed number of units, located near a home base (post), and a flag. Units have a range in which they can attack an opponent. A successful attack reduces the nearby enemy unit's hit-points. When a unit's hit-points drops to zero, the unit is "dead" and removed from the game.

The objective of CTF is to capture the opponent's flag. Each unit has the ability to pick-up or drop the enemy flag. To win the game, the flag must be picked up, carried, and dropped at the friendly home base. If a unit is killed while carrying the flag, the flag is dropped at the unit's location, and can later be picked up by another unit. A unit cannot pick up its own side's flag at any time, as this would make it too easy to protect the flag.

Terrain is very important to CTF. For most of our tests we keep it simple and symmetric to avoid bias towards either side. However, even with more complex terrains, while there may be a bias towards one side, it is expected that planners that perform better on symmetric maps will also perform better on complex maps. While CTF does not capture all the elements involved in a full RTS game — such as economy and army-building — it is a good scenario for testing planning algorithms. Many of the features of full RTS games are present in CTF — including scouting and base defense.

Before we discuss how we applied MCPlan to a CTF game we first describe the simulation software we used.

3.3.1 ORTS

ORTS (Open RTS) is a free software RTS game engine which is being developed at the University of Alberta and licensed under the GNU General Public License. The goal of the project is to provide AI researchers and hobbyists

with an RTS game engine that simplifies the development of AI systems in the popular commercial RTS game domain. ORTS implements a server–client architecture that makes it immune to map–revealing client hacks which are a widespread plague in commercial RTS games. ORTS allows users to connect *whatever* client software they wish — ranging from distributed RTS game AI to feature–rich graphics clients. The CTF game which we use for studying MCPlan performance has been implemented within the ORTS framework. For more information on the status and development of ORTS we refer readers to [8][13].

3.3.2 CTF Game State Abstraction

In the state representation, the map is broken up into tiles (representing a set of possible unit locations). Units are located on these tiles, and their positions are reasoned about in terms of tiles, rather than exact game coordinates. The state also contains information about the units' hit–points, as well as locations of walls and flags.

3.3.3 Evaluation Function

We tried to keep our evaluation function simple and obvious, without relying on a lot of expert knowledge. The evaluation function for our CTF AI has three primary components: material, exploration/visibility, and capture/safety. The first two components are standard to any RTS game. The third component is specific to our CTF scenario. Without it, the AI would have no way to know that it was actually playing a CTF game, and it would behave as if it was a regular battle. In each component the difference of the values for both players is computed. In the following we briefly give details of the evaluation function.

Material

The most important part of any RTS game is material. In most cases, the side with the most resources — including military units, buildings, etc. — is the victor. Thus, maximizing material advantage is a good sub–goal for any planning AI. This material can later be converted into a decisive advantage such as having a big enough army to eliminate the enemy base. There is a question of how to compare healthy units to those with low hit–points. For example, while it may be clear that two units each with 50% health are better than one unit with 100% health, which would be better, one unit with 100% health, or two units with 25% health? While the two units could provide more firepower, they could also be more quickly killed by the enemy. There are different situations where the values of these units may be different. For our tests, we provide a simple solution: each unit provides a bonus of $\sqrt{0.01 \times \text{hp}}$. The maximum hp (hit–point) value is 100. Thus, each live unit has a value of between 0.1 and 1. The value for friendly units is added to our evaluation, and enemy units values are subtracted. Taking the square root prefers states which — for a constant hit–point total — have a more balanced hit–point distribution.

Exploration and Visibility

When not doing something of immediate importance, such as fighting, exploring the map is very important. The side with more information has a definite advantage. Keeping tabs on the enemy, finding out the lay of the land, and discovering the location of obstacles are all important. The planner cannot accurately evaluate its plans unless it has a good knowledge of the terrain and of enemy forces and their locations. The value of information is reflected by these evaluation function sub–components:

- Exploration bonus: 0.001× # of explored tiles, and

- Vision bonus: 0.0001× # visible tiles.

Note that the bonus values can be changed or even learned.

Flag Capture and Safety

To win a CTF game, the opponent's flag has to be captured.

It is important to encourage the program to go after the enemy's flag, while at the same time ensuring that the program's flag remains safe:

- Bonus for being close to enemy flag: +0.1 per tile,
- Bonus for possession of enemy flag: +1.0,
- Bonus for bringing enemy flag closer to our base: +0.2 per tile, and
- Similar penalties apply if the enemy meets these conditions.

Note that all these heuristic values have been manually tuned. Machine learning would be a way to more reliably set these values.

Combining the Components

The simplest thing to do, and what we do right now, is have constant factors for adding the three components together. There are exceptions where this is not the best approach. For example, if we are really close to capturing the enemy flag, we may choose to ignore the other components, such as exploration. Such enhancements are left as future work.

Evaluation Function Quality

We can perform experiments to test the effectiveness of our evaluation function. For example, we could measure the time it takes to capture the flag if there are no enemy units. This removes all tactical situations and focuses on testing that the evaluation function is correctly geared towards capturing the enemy flag. Playing the MCPlan AI against a completely random AI also provides a good initial test of the evaluation function. A random evaluation function

would perform on the same level as the random AI, whereas a better evaluation function would win more often.

3.3.4 Plan Generation

There are two types of plan generation used in this project: random and scripted. The random plans are simple and are described below. The scripted plans are slightly more sophisticated, but still quite simple.

Random Plans

A random plan consists of assigning a random nearby destination for each unit to move to. That is, for each unit, a nearby unoccupied destination tile is selected. The maximum distance to the destination is determined by the `max_dist` variable. The A* pathfinding algorithm is then used to find a path for each unit. Note that collisions are possible between the units, but are ignored for planning purposes. We did not implement any group–based pathfinding, although it is a possible enhancement.

Scripted Plans

We have implemented a small number of action scripts which provide test opponents for the MCPlan algorithm. As previously mentioned, scripted plans have many disadvantages — most notably, the need to have an expert define, refine and test them. However, there is the possibility that given a set of scripted plans, applying the search and simulation algorithms described in this paper can result in a stronger player.

3.3.5 Plan Step Simulation

Simulations must be used because when the planner evaluates an action, the result of that action cannot be perfectly determined because of hidden enemy units, unknown enemy actions, randomized action effects, etc. Also, as our

simulation acts on an abstracted state description, the computation should be fast. The plan step simulation function takes the given plans for the friend and enemy sides and executes one–tile moves for each side. Unit attacks are then simulated by selecting the nearest opposing unit for each unit, and reducing its hit–points. The attacks may not match what would happen in the actual game, due to many reasons. For example, units may seem to be in range but actually they are not, due to the abstracted distances. Also, in some games, the attack damage is random, so the damage results may not be exactly the same as what will happen in the game. However, it is expected that with a large enough value of `num_evals`, the final result should be more statistically accurate.

3.3.6 Other Issues

In this subsection we discuss some implementation issues related to developing and testing a search/simulation based RTS planning algorithm such as MCPlan.

Map Generation

It is clear that in performing the tests, map generation is a hard problem. To produce an unbiased map, the map should be completely symmetric. A more complex asymmetric map could favor one side. In addition, it is possible that different types of maps could favour different AIs. For our tests we use a simple symmetric map, to avoid most of these issues. It is expected — and to be confirmed — that on more complex and on randomly generated maps, the conclusions we draw from our experiments should still hold.

Server Synchronization

The tests should be run with server synchronization turned on. This option tells the ORTS server to wait for replies from both clients before continuing on

to the next turn. In the default mode with synchronization off, the first player to connect may possibly have an advantage, due to being able to move while the second player's process is still initializing its GUI, etc. The server synchronization option eliminates this possible source of bias, as well as reducing the randomness caused by random network lag.

Interactions and Replanning

As players interact, previous planning may quickly become irrelevant. In many cases, replanning must occur. Not every interaction should result in replanning. This would result in too frequent replanning, which would slow down the computation while perhaps not improving the decision quality much. Instead, only important interactions should result in replanning. Possible such interactions are: "a unit is destroyed," "a unit is discovered," or "a flag is picked up." Note that attacks, while important, happen too frequently and thus should not trigger replanning.

3.4 Summary

The typical expert–knowledge–based system in current commercial RTS games places a huge burden on the developers and testers, and is unable to adapt to new situations. We have described a plan selection algorithm – MCPlan – which is based on Monte–Carlo sampling, simulations, and replanning. This reduces the amount of expert–knowledge needed in an RTS planner, and instead relies on computation. A simple CTF scenario is used to demonstrate MCPlan.

Chapter 4

Experiments

In this chapter, we investigate the performance issues of MCPlan on our CTF game.

4.1 Experimental Design

Each experimental data point consisted of a series of games between two CTF programs. The experiments were performed on 1.2 GHz PC's with 1 GB of RAM. Note that because the experiments were synchronized by the ORTS server the speed of the computer does not affect the results. Each data point is based on the results of matching two programs against each other for 200 games. For a given map, two games are played with the programs playing each side once. A game ends when a flag is captured, or one side has all their men eliminated. A point is awarded to the winning side. Draws are handled depending on the type of draw. If the game times out and there is no winner, then neither side gets a point. If both sides achieve victory at exactly the same time, then both sides get a point. The reported win percentage is one side's points divided by the total points awarded in that match. In a match with no draws the total points is equal to the number of games (200).

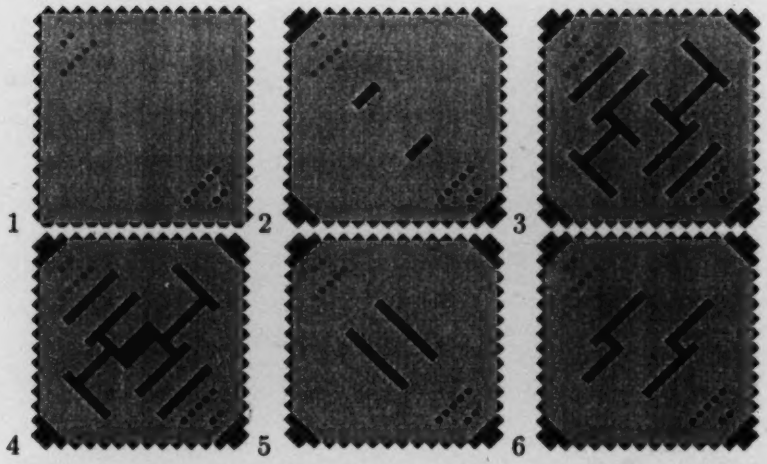

Figure 4.1: Maps and unit starting positions used in the experiments: **map 1** (upper left): empty terrain (this is the default), **map 2**: simple terrain with a couple of walls, **map 3**: complex terrain, **map 4**: complex terrain with dead–ends, **map 5**: simple terrain with a bottleneck, **map 6**: intermediate complexity.

4.1.1 Maps

Figure 4.1 shows the maps that have been used in the experiments. Their dimensions are 20 by 20 tiles. By default each side starts with five men.

4.1.2 Search Parameters

The `max_dist` parameter is the maximum distance that a unit can move from its current position in a randomly generated plan. In all these experiments, the `max_dist` parameter is set to 6 tiles, unless otherwise stated. The unit's sight radius is set to 10 tiles, and unit's attack range is set to 5 tiles. To reduce the number of experiments needed, the number of simulations (`num_sims`) is set to be equal to the number of plans (`num_plans`). This makes sense as the number of simulations is also the number of opponent plans considered.

4.1.3 Players

There are two opponents tested in these experiments other than the MCPlan player: Random and Rush-the-Flag. Random is equivalent to MCPlan running with `num_plans` = 1. It simply generates and executes a random plan, using the same plan generator as the MCPlan player. Random is still a reasonable strong player, as it automatically attacks nearby enemy units. Its strategy is like giving 5 men guns, and telling them to run around randomly, killing any enemies sighted, and picking up the enemy flag if they happen to encounter it. In other words, it is still a dangerous opponent.

Rush-the-Flag is a scripted opponent which behaves as follows:

1. If the enemy flag is not yet captured, send all units towards the enemy flag and attempt to capture it.

2. If the enemy flag is captured, have the flag carrier return home. All other units follow the flag carrier.

While simple in design, and easily implemented in about 50 lines of code, the Rush-the-Flag opponent proves to be a strong adversary.

Stand-Still is a scripted opponent which only stands still and attacks nearby enemy units. As the units initially start in a line formation near the friendly flag, it is a very defensive player. While it never captures the enemy flag, it can still win games if the enemy units are all destroyed.

4.1.4 Issues

In order to make sure that there was no first player advantage (or disadvantage) server synchronization was turned on. In the resulting experiments, the first player wins very close to 50% the time, as expected.

The evaluation function parameters were hand-tuned to maximize win percentage and minimize the time it takes to capture the flag and return it,

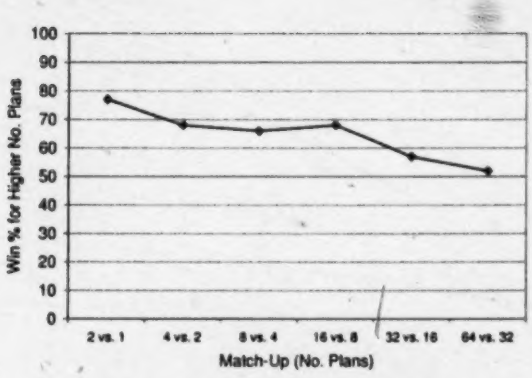

Figure 4.2: Increasing Number of Plans

with weak or no opposition. The same evaluation function, as described in chapter 3, is used for all these experiments.

4.2 Results

We now investigate the performance of MCPlan against a variety of opponents and using different combinations of search parameters.

4.2.1 Increasing Number of Plans

In Figure 4.2, the performance of the MCPlan algorithm on the default map is evaluated as a function of the number of plans considered. Each data point represents the result of a player considering p plans playing against one that considers $2p$ plans. This results show that the program's play improves as the number of plans increases, but with diminishing returns. Eventually, the sample size is large enough that adding more plans results in marginal performance improvements, as expected.

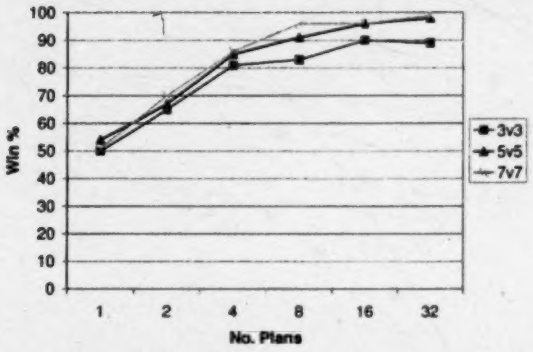

Figure 4.3: Different Number of Units. MCPlan vs. Random

4.2.2 Number of Units

Figure 4.3 shows the results when the number of units is varied. The results in the figure are for MCPlan against Random on the default map. As expected, regardless of the number of units aside, increasing the number of plans improves the performance of the MCPlan player. With a larger number of units per side, MCPlan wins more often. This is reasonable, as the number of decisions increases with the number of units, and there is more opportunity to make "smarter" moves.

4.2.3 Different Maps

The previous results were obtained using the same map (empty map, the default). Do the results change significantly with different terrain? In this experiment, we repeat the previous matches using a variety of maps. Figure 4.4 shows the results. Note that one map has 7 men aside. The results indicate that MCPlan is a consistent winner, but the winning percentage depends on the map. The more complex the map, the better the random player performs. This is reasonable, since with more walls, there is more uncertainty as to where enemy units are located. This reduces the accuracy of the MCPlan player's simulations. In the tests using the map with a bottleneck (map 5), the

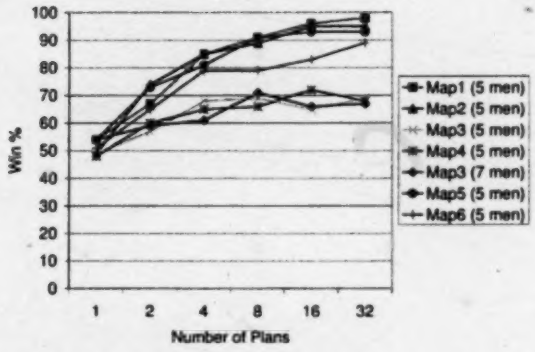

Figure 4.4: Different Maps. MCPlan vs. Random

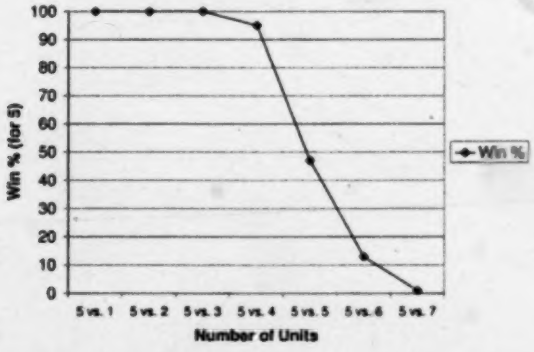

Figure 4.5: Unbalanced Number of Units and Same AI

performance was similar to the tests with simple maps without the bottleneck. This shows that the simulation is capable of dealing with bottlenecks, at least in simple cases.

4.2.4 Unbalanced Number of Units

Figure 4.5 shows that in games with an unbalanced number of units, the side with more men wins more often. This experiment is with MCPlan versus MCPlan. The number of plans is set to 8 for both sides. This is a control case for the next experiment.

Figure 4.6 illustrates the relative performance between MCPlan and Ran-

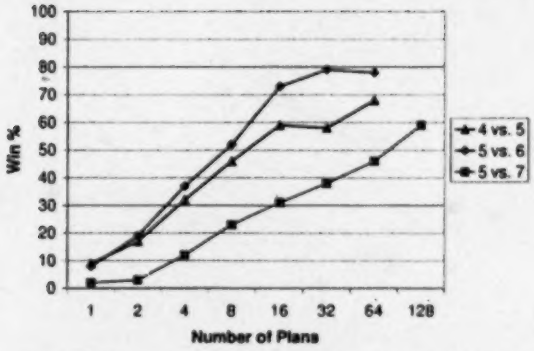

Figure 4.6: Less Men and Stronger AI vs. Random

dom when Random is given more men. The results show that given a sufficient number of plans to evaluate, MCPlan with less men and better AI can overcome Random with more men but a poorer AI. The results suggest that using MCPlan is strong enough to overcome a significant material advantage possessed by the weaker AI (Random). The figure shows the impressive result that 5 units with smart AI defeat 7 units with dumb AI 60% of the time when choosing between 128 plans.

4.2.5 Optimizing Max–Dist

A higher `max_dist` value results in longer plans, which allows more look–ahead, as well as a higher number of possible plans. The higher number of possible plans may increase the number of plans required to find a good plan.

More look–ahead should help performance. However, with too much look–ahead, noise may become a problem. The noise is due to errors in the simulation — which uses an abstracted game state — and incorrect predictions of the opponent plan. The longer we need to guess what the opponent will do, the more likely we are to make an error. So, more simulations are required to have a good chance of predicting the opponent's plan or something close enough to it.

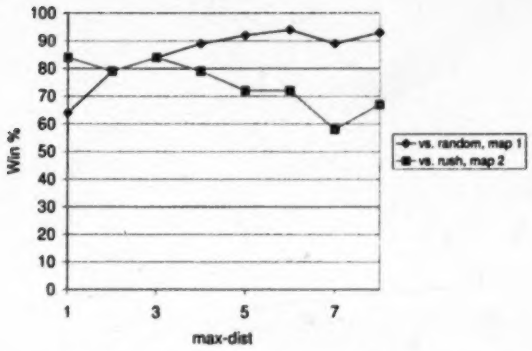

Figure 4.7: Optimizing Max-Dist Parameter

In this experiment we vary the `max_dist` parameter to optimize the win percentage against the Random opponent on map 1 and the Rush-the-Flag opponent on map 2 (see Figure 4.7). The planner playing against random achieves its best performance of 94% at dist=6. Note that although one may expect MCPlan to score 100% against Random, in practice this will not happen. A lone unit may unexpectedly encounter a group of enemy units. Once engaged in a losing battle, it is difficult to retreat, since all units move at the same speed. The performance of MCPlan vs. Rush-the-Flag becomes worse as `max_dist` is increased. This is due to the incorrect predictions of the opponent plan, as we are generating opponent plans randomly and not correctly anticipating Rush-the-Flag's strategy. However, this should be at least partially handled by increasing the number of plans, as shown in the next experiment.

4.2.6 Scripted Opponents

Rush-the-Flag Opponent

Figure 4.8 shows MCPlan playing against Rush-the-Flag. The playing strength of Rush-the-Flag is very map dependent, as it has a fixed strategy. On the first map, Rush-the-Flag wins nearly every game. Rushing is a near-

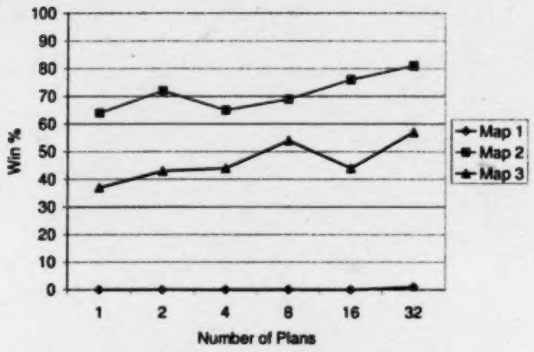

Figure 4.8: MCPlan vs. Rush-the-Flag Opponent

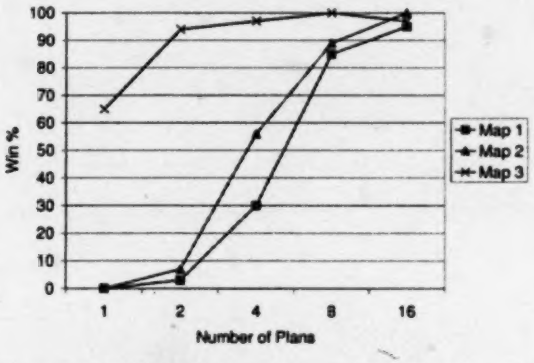

Figure 4.9: MCPlan vs. Stand-Still Opponent

optimal strategy on an empty map. On map 2, where the direct path to the other side is blocked, Rush-the-Flag is much weaker. MCPlan wins more than 60% of the time even with `num_plans`=1. With `num_plans`=32, MCPlan wins more than 80% of the time. However, on map 3, where the map is more complex and all paths to the other side are long, Rush-the-Flag again becomes a challenging opponent. However, with `num_plans`=32, MCPlan wins more than 55% of the games.

Stand-Still Opponent

The Stand-Still opponent is stronger than the completely random opponent on some maps. In the starting positions, the units are grouped up in a line

formation and all protecting the flag. On the complex map, since the random opponent can only approach from the sides, Stand-Still is easily defeated, even with `num_plans=1`. However on the simpler maps (1 and 2) where the area in front of the starting units is open, a random opponent is easily defeated by Stand-Still (100% of the time). In all cases, as the number of plans is increased, MCPlan begins to win consistently against Stand-Still, eventually defeating Stand-Still nearly 100% of the time.

4.2.7 Run-Time for Experiments

In order to get more statistically valid results, the experiments were not run in real-time. Rather, they were run much faster than real-time, about 100 times faster. This allowed us to run more games, resulting in more statistically meaningful results.

While the run-time depends on the parameters, using typical parameters (map 2, 16 plans, 5 men per side) a 200-game match runs in about 80 minutes on our test machines. The average time per game is less than 30 seconds. As the planner re-plans hundreds of times per game, this results in planning times of a fraction of a second. Note that increasing the number of plans increases the run-time quadratically, since both the number of friendly plans and opponent plans is increased.

4.3 Conclusions

The performance of MCPlan has been tested against a few different opponents and using different combinations of search parameters. Increasing the number of plans is shown to improve the program's play, but with diminishing returns. Performance is better with a larger number of units and simpler maps. MCPlan is able to win consistently against a random player that is given more units. While these are simple scenarios, MCPlan only takes a fraction of a

second to replan, which should allow it to handle more sophisticated RTS scenarios.

Chapter 5

Conclusions and Future Work

This thesis has presented preliminary work in the area of sampling–based planning in RTS games. We have described a plan selection algorithm – MCPlan – which is based on Monte–Carlo sampling, simulations, and replanning. Applied to simple CTF scenarios MCPlan has shown promising initial results. The strength of plans is automatically evaluated, allowing the AI to select the strongest of a set of plans. This process is automatic and relies on computation rather than expert knowledge. While the plans used in these experiments were completely randomly generated, this does not have to be the case. Using MCPlan with a set of well–scripted plans should result in an AI that is stronger than using any individual fixed plan, or randomly selecting a fixed plan. MCPlan requires running many simulations to achieve good performance. This may in some cases be too time consuming, although our experiments do suggest that it is feasible. In some cases a simulator may be difficult to write. Since RTS games are already simulations, this should not be a problem. To gauge the true potential of MCPlan we need to compare it against a highly tuned scripted AI, which was not available at the time of writing. In addition we need to test MCPlan against human opponents. We intend to extend MCPlan in various dimensions and apply it to more complex RTS games. For instance, it is natural to add knowledge about opponents in the form of

plans that can be incorporated in the simulation process to exploit possible weaknesses. Also, the top–level move decision routine of MCPlan should be enhanced to generate move distributions rather than single moves which is especially important in imperfect information games. Lastly, applying MCPlan to bigger RTS game scenarios requires us to consider more efficient sampling and abstraction methods.

Bibliography

[1] http://www.blizzard.com.

[2] http://www.ensemblestudios.com.

[3] ftp.eecs.umich.edu/~soar/tutorial.html.

[4] ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z.

[5] http://www.empireearth.com.

[6] http://www.ensemblestudios.com/news/devnews/terrain1.shtml.

[7] http://www.ict.usc.edu/disp.php?bd=proj_games_fsc1.

[8] http://www.cs.ualberta.ca/~mburo/orts.

[9] B. Abramson. Expected-outcome: a general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:182–193, 1990.

[10] D. Billings, L. Pena, J. Schaeffer, and D. Szafron. Using probabilistic knowledge and simulation to play poker. In *AAAI National Conference*, pages 697–703, 1999.

[11] B. Bouzy. Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. In *Joint Conference on Information Sciences*, pages 505–508, 2003.

[12] B. Bouzy and B. Helmstetter. Monte Carlo Go developments. In *Advances in Computer Games X*, pages 159–174. Kluwer Academic Press, 2003.

[13] M. Buro and T. Furtak. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS), Arlington VA 2004*, pages 51–58, 2004.

[14] E. Dybsand. Goal-directed behavior using composite tasks. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 237–245. Charles River Media, 2004.

[15] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[16] I. Frank and D.A. Basin. Search in games with incomplete information: A case study using bridge card play. *Artificial Intelligence*, 100(1-2):87–123, 1998.

[17] M. Ginsberg. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*, pages 584–589, 1999.

[18] M. Grimani. Wall building for RTS games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 425–437. Charles River Media, 2004.

[19] T. Kent. Multi-tiered AI layers and terrain analysis for RTS games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 447–455. Charles River Media, 2004.

[20] J. J. Lee. *A Simulation-Based Approach for Decision Making and Route Planning.* PhD thesis, University of Florida, August 1996.

[21] J. J. Lee and P. A. Fishwick. Real-time simulation-based planning for computer generated force simulation. *Simulation*, 63(5):299–315, November 1994.

[22] L. Liden. Strategic and tactical reasoning with waypoints. In Steve Rabin, editor, *AI Game Programming Wisdom*, pages 211–220. Charles River Media, 2002.

[23] J. Orkin. Applying goal-oriented action planning to games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 217–227. Charles River Media, 2004.

[24] M. Ramsey. Designing a multi-tiered AI framework. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 457–466. Charles River Media, 2004.

[25] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 2nd edition, 2003.

[26] J. Schaeffer, D. Billings, L. Peña, and D. Szafron. Learning to Play Strong Poker. In J. Fürnkranz and M. Kubat, editors, *Machines That Learn To Play Games*, pages 225–242. Nova Science Publishers, 2001.

[27] B. Sheppard. *Towards Perfect Play in Scrabble.* PhD thesis, University of Maastricht, 2002.

[28] B. Sheppard. Efficient control of selective simulations. *Journal of the International Computer Games Association*, 27(2):67–80, 2004.

[29] S. Shoemaker. Random map generation for strategy games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 405–412. Charles River Media, 2004.

[30] R. Sutton and A. Barto. *Reinforcement Learning.* MIT Press, 1998.

[31] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[32] P. Tozour. Influence mapping. In M. Deloura, editor, *Game Programming Gems 2*, pages 287–297. Charles River Media, 2001.

[33] P. Tozour. Using a spatial database for runtime spatial analysis. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 381–390. Charles River Media, 2004.

[34] N. Wallace. Hierarchical planning in dynamic worlds. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 229–236. Charles River Media, 2004.

[35] S. Willmott, J. Richardson, A. Bundy, and J. Levine. Applying adversarial planning techniques to Go. *Theoretical Computer Science*, 252(1–2):45–82, 2001.